

Algoritmos y Estructuras de Datos II

Segundo parcial — 14 de junio de 2014

Aclaraciones

- El parcial es **a libro abierto**.
- Cada ejercicio debe entregarse en **hojas separadas**.
- Incluir en cada hoja el número de orden asignado, número de hoja, apellido y nombre.
- Al entregar el parcial, completar el resto de las columnas en la planilla.
- Cada ejercicio se calificará con **MB**, **B**, **R** o **M**, y podrá recuperarse independientemente de los demás. Podrán aprobarse los parciales de la materia con hasta 2 (dos) ejercicios **R** (regular) **entre ambos exámenes**, siempre que ninguno de ellos sea un ejercicio 1. Para más detalles, ver “Información sobre la cursada” en el sitio Web.

Ej. 1. Diseño

El siguiente TAD especifica el comportamiento del servidor de la nueva red social “ReSocial”. En “ReSocial”, los miembros se dedicarán exclusivamente a “megustearse” unos a otros. Pueden registrarse nuevos miembros en todo momento, y se quiere saber siempre cuántos “me gusta” ha recibido cada miembro.

Cada tanto los miembros deciden expulsar al miembro menos popular de la red (que es alguno de los que menos “me gusta” tienen). Las personas que han sido echadas no pueden volver a ser miembros.

TAD PERSONA ES NAT

TAD RESOCIAL

generadores

crear	:		→	servidor	
registrar	:	servidor $s \times$ persona p	→	servidor	$\{p \notin \text{miembros}(s) \wedge p \notin \text{echados}(s)\}$
recibirMeGusta	:	servidor $s \times$ persona p	→	servidor	$\{p \in \text{miembros}(s)\}$
echarAlMenosPopular	:	servidor s	→	servidor	$\{\text{miembros}(s) \neq \emptyset\}$

observadores básicos

miembros	:	servidor	→	conj(persona)	
echados	:	servidor	→	conj(persona)	
puntaje	:	servidor $s \times$ persona p	→	nat	$\{p \in \text{miembros}(s)\}$

otras operaciones

miembrosConEstePuntaje	:	servidor $s \times$ nat	→	conj(persona)	
mínimoPuntaje	:	servidor s	→	nat	$\{\text{miembros}(s) \neq \emptyset\}$
menosPopular	:	servidor s	→	persona	$\{\text{miembros}(s) \neq \emptyset\}$

axiomas ...

menosPopular(s) \equiv dameUno(miembrosConEstePuntaje(s , mínimoPuntaje(s)))

...

Fin TAD

Diseñe el tipo abstracto de datos RESOCIAL respetando los siguientes requerimientos:

- En el diseño, como criterio de desempate se elegirá a quien tenga menor identificador de persona. Es decir que MENOSPOPULAR(s) deberá devolver la mínima persona entre aquellas que tengan el mínimo puntaje.
- Los requerimientos de complejidad temporal en **peor caso** son:
 - RECIBIRMEGUSTA(s, p) debe resolverse en $O(\log m_s)$
 - MENOSPOPULAR(s) debe resolverse en $O(\log m_s)$
 - ECHARALMENOSPOPULAR(s) debe resolverse en $O(\log m_s)$
 - ECHADOS(s) debe resolverse en $O(1)$

siendo $m_s = \#(\text{miembros}(s))$, es decir la cantidad de miembros **actuales** (no echados) del servidor s .

- a) Muestre la estructura de representación propuesta.
Explique para qué sirve cada parte, o utilice nombres autoexplicativos.
- b) Escriba en pseudocódigo los 4 algoritmos para los que se piden requerimientos de complejidad.

Justifique claramente cómo y por qué los algoritmos, la estructura y los tipos soporte permiten satisfacer los requerimientos pedidos. No es necesario diseñar los módulos soporte, **pero sí describirlos, justificando por qué pueden (y cómo logran)** exportar los órdenes de complejidad que su diseño supone.

Ej. 2. Dividir y conquistar

Un árbol binario es *bicolor* si sus nodos son de color rojo o negro.

Un árbol binario bicolor es *rojinegro válido*¹ si y sólo si satisface las siguientes 3 propiedades:

1. Todas las hojas son negras.
2. Los hijos de un nodo rojo, de haberlos, siempre son negros.
3. Todos los caminos (desde la raíz hasta una hoja) contienen la misma cantidad de nodos negros.

Escriba un algoritmo *EsRojinegroVálido?*, basado en la técnica de dividir y conquistar, que dado un árbol binario bicolor determine si es o no rojinegro válido. No visite el mismo nodo múltiples veces. Marque claramente en su algoritmo las distintas etapas de la técnica.

Suponga que puede determinar en $O(1)$ si un nodo es rojo o negro.

Calcule la complejidad temporal de su algoritmo en peor caso. Justifique.

Ej. 3. Ordenamiento

Un melómano quiere ordenar su gigantesca colección de mp3 de manera tal que quede ordenada por nombre del artista, desempataando por nombre del disco y finalmente por nombre del tema.

Los discos y los temas siempre tienen nombres conformados únicamente por caracteres estándar: dígitos y letras del alfabeto latino sin acentos. Sin embargo, los artistas pueden tener nombres raros como Björk, Ƿ, BøøM!, MëgaŽaurus, 000, etcétera. No sabemos cuántos caracteres raros existen, pero sí ordenarlos.

Afortunadamente sabemos que los nombres de bandas tienen a lo sumo 40 caracteres; no así los nombres de discos y temas, que pueden tener cualquier longitud.

Se pide ordenar la colección de mp3 en $O(n \cdot \log(b) \cdot |d| \cdot |t|)$, donde n es la cantidad de mp3s, b la cantidad de bandas distintas, $|d|$ la máxima longitud de nombre de disco y $|t|$ la máxima longitud de nombre de tema.

Justifique por qué su solución cumple lo pedido. Puede combinar pseudocódigo y/o descripción en castellano según considere conveniente. Procure que su justificación sea clara y legible, y no omita premisas relevantes.

¹La definición que usamos en este ejercicio es una versión simplificada de los *red-black trees* clásicos.

1) a) ReSocial se representa con estr, donde estr es:
Tupla < Puntajes: conj(nat), miembrosPuntaje: dicc(nat, conj(persona)),
MG: dicc(persona, nat), echados: conj(persona) >

• En Puntajes se van a guardar todos los puntajes para los cuales algún miembro tiene ese puntaje (no va a haber miembros que tengan como puntaje un número que no esté ahí). Notar que, en el peor de los casos, va a haber tantos Puntajes como miembros, así que podemos acotar #Puntajes por ms ✓

• En miembrosPuntaje, dado un puntaje, obtenemos el conjunto de miembros con ese puntaje. Las claves deben ser las mismas que aparecen en "Puntajes", por lo que no puede haber "significados varios" (conjunto vacío no está en los Significados). y, otra vez, las claves están acotadas por ms.

• En MG, dada una persona, obtenemos la cantidad de me gusta que tiene. Notar que entonces todos los Significados de "MG" pertenecen a "Puntajes" y son, por ende, claves de "miembrosPuntaje".

• En echados se "guardan" los miembros que fueron echados. Estos no pueden ser claves de "MG" ni pertenecer a ningún Significado de "miembrosPuntaje".

b) Recibir Me Gusta (in/out s: estr, in p: persona)

nat viejo ← obtener(p, s.MG)

$O(\log(ms))$

nat nuevo ← viejo + 1

$O(1)$

definir(p, nuevo, s.MG)

$O(\log(ms))$

conj(persona) persPunt ← obtener(viejo, s.miembresPunt)

$O(\log(ms))$

eliminar(p, persPunt)

$O(\log(ms))$

if vacío?(persPunt)

$O(1)$

borrar(viejo, s.miembresPunt)

$O(\log(ms))$

eliminar(viejo, s.puntajes)

$O(\log(ms))$

if definido?(nuevo, s.miembresPunt)

$O(\log(ms))$

conj(persona) puntNuevo ← obtener(nuevo, s.miembresPunt)

$O(\log(ms))$

Ag(p, puntNuevo)

$O(\log(ms))$

else

conj(persona) puntNuevo ← vacío()

$O(1)$

Ag(p, puntNuevo)

$O(\log(ms))$

definir(nuevo, puntNuevo, s.miembresPunt)

$O(\log(ms))$

— 0 —

i: Echar A Menos Popular (in/out s: estr)

itA ← crearIT(s.puntajes)

// Asumo que el Conj tiene un iterador que puede crear en

$O(\log(ms))$

conj(persona) menosPopu ← obtener(siguiente(itA), s.miembresPuntaje)

$O(\log(ms))$

itB ← crearIT(menosPopu)

$O(\log(ms))$

persona aEchar ← siguiente(itB)

$O(1)$

Ag(aEchar, s.echados)

$O(\log(ms))$

eliminar(aEchar, menosPopu)

$O(\log(ms))$

if vacío?(menosPopu)

$O(1)$

borrar(siguiente(itA), s.miembresPuntaje)

$O(\log(ms))$

eliminarSiguiente(itA)

$O(\log(ms))$

bonar (aEchar, s. MG) $O(\log(m_s))$

— o —

i MenPopular (in s: esr) \rightarrow res: person

itA \leftarrow crearIt(s. Puntaje) $O(\log(m_s))$

conj(persona) menorPopu \leftarrow obtener(siguiente(itA), s. miembrosPuntaje)

itB \leftarrow crearIt(menorPopu) $O(\log(m_s))$

res \leftarrow siguiente(itB) $O(1)$

— o —

i Echados (in s: esr) \rightarrow res: conj(persona)

res \leftarrow s. echados. $O(1)$

— o —

Para cumplir las complejidades, se necesita:

- Los conjuntos deben tener complejidad $O(\log(n))$ para decir si un elemento pertenece o no, para agregar un elemento, y para eliminarlo, donde n representa la cantidad de elementos del conjunto. "Vacio?" debería tener complejidad $O(1)$. Además, su iterador debería crear "avanzando" al índice ^{mínimo} del conjunto como siguiente.

Todo esto puede lograrse si se utiliza un AVL, y su iterador se crea ($\text{en } O(\log(n))$) avanzando al elemento más chico. Como en un AVL esto es ir todo lo pare se puede hacer abajo y a la izquierda, esto está notado por su altura, que es $\log(n)$. La op. "eliminarSiguiente" entonces tampoco tardaría más que \log .

-) El diccionario debería contar con las operaciones definidas?, definir, obtener y borrar en $O(\log(n))$.
Otra vez. Si se implementa sobre AVL todas estas operaciones cumplen la complejidad.

2) Es Rojo Negro Válido? ($ab: A$) \rightarrow bool : res

res $\leftarrow \pi_1(\text{verificar}(A))$ $O(1)$ * complejidad de verificar(A)

Verificar ($ab: A$) \rightarrow res: $\langle \text{bool}, \text{nat}, \text{color} \rangle$

if nil?(A)

res $\leftarrow \langle \text{true}, 0, \text{negro} \rangle$

else if esHoja?(A)

~~res $\leftarrow \langle \text{color}(A) == \text{negro}$~~

if color(A) == negro

| res $\leftarrow \langle \text{true}, 1, \text{negro} \rangle$

else

| res $\leftarrow \langle \text{false}, 0, \text{rojo} \rangle$

else if soloHijoIzq(A)

~~ab~~ ver $\leftarrow \text{Verificar}(\text{izq}(A))$

if color(raiz(A)) == negro

| res $\leftarrow \langle \pi_1(\text{ver}), \pi_2(\text{ver}) + 1, \text{negro} \rangle$

else

| res $\leftarrow \langle \pi_1(\text{ver}) \wedge \pi_3(\text{ver}) == \text{negro}, \pi_2(\text{ver}), \text{rojo} \rangle$

else if soloHijoDer(A)

ab ver $\leftarrow \text{Verificar}(\text{der}(A))$

if color(raiz(A)) == negro

| res $\leftarrow \langle \pi_1(\text{ver}), \pi_2(\text{ver}) + 1, \text{negro} \rangle$

else

| res $\leftarrow \langle \pi_1(\text{ver}) \wedge \pi_3(\text{ver}) == \text{negro}, \pi_2(\text{ver}), \text{rojo} \rangle$

Fíjate que entre las dos ramas del IF está la línea

else

ab verI \leftarrow verificar(izq(A))

ab verD \leftarrow verificar(der(A))

if color(raiz(A)) == negro

res $\leftarrow \pi_1(\text{verI}) \wedge \pi_1(\text{verD}) \wedge \pi_2(\text{verI}) = \pi_2(\text{verD}), \pi_2(\text{verI}) + 1, \text{negro}$

else

res $\leftarrow \langle \pi_3(\text{verI}) == \text{negro} \wedge \pi_3(\text{verD}) == \text{negro} \wedge \pi_1(\text{verI}) \wedge \pi_1(\text{verD}) \wedge \pi_2(\text{verI}) = \pi_2(\text{verD}), \pi_2(\text{verI}), \text{rojo} \rangle$

• En peor caso, el árbol va a estar degenerado a izquierda o derecha, y por cada nodo, en el algoritmo "Verificar" hago todas operaciones $O(1)$ (comparaciones, creación de tuples, sumas). Entonces, por cada llamado recursivo voy a tener $n-1$ elementos.

Luego, la complejidad de verificar (y por consiguiente, de EsRojoNegroVálido?) es $O(n)$, donde n es la cant. de nodos del árbol.

- EsRojo?(A) \rightarrow bool

~~return izq(A) == nil \wedge der(A) == nil~~

res \leftarrow nil?(izq(A)) \wedge nil?(der(A))

- SoloHijoIzq(A) \rightarrow bool

res \leftarrow nil?(der(A)) \wedge \neg nil?(izq(A))

- SoloHijoDer(A) \rightarrow bool

res \leftarrow nil?(izq(A)) \wedge \neg nil?(der(A))

3) Asumo que tengo implementaciones de diccionarios sobre trie, y sus respectivos iteradores, que funciona de la forma esperada: recorren las cadenas en orden lexicográfico (respecto de sus claves).

~~Para empezar hago una recolección~~

Además, pienso la colección como un arreglo de "MP3", donde MP3 son tuplas <artista, disco, tema>.

~~Primero, hago una pasada en $O(n)$ sobre el arreglo original.~~
~~Fin de la pasada~~

También asumo que tengo implementaciones de diccionarios sobre AVL, también con los iteradores esperados: recorren las entradas de acuerdo a la relación de orden ^{ascendente} que hay entre sus claves.

Primero, creo un diccionario ^{vacio} sobre AVL, con claves de tipo "banda" y significados $\text{dicc}(\text{disco}, \text{dicc}(\text{tema}, \langle \text{MP3}, \text{not} \rangle))$ estando estos dos últimos diccionarios implementados sobre trie.

Luego, recorro el arreglo original (en $\Theta(n)$) y por cada MP3 "x", me fijo su banda. Si no está definida en el dicc de banda, la defino primero (sino, no), con signif "vacio". Luego, obtengo su signif. y por defino para el primer diccionario (el de disco), una entrada de clave k. disco, y signif. vacío. Luego, una vez definido,

obtengo su signif (el dicc. de "tema") y, nuevamente, me fijo si está definido k. tema. Si está, incremento en 1 el nro de signif, y paso al siguiente mp3 en el arreglo original. Sino, lo defino con $\langle k, 1 \rangle$, como signif.

Hacer esto lleva $O(\log(b) + |d| + |T|)$ para cada mp3, así que en total cuesta $O(n(\log(b) + |d| + |T|))$.

Cuando termine, itero primero sobre el AVL, y para cada signif., itero sobre el dicc de disco, y para cada uno de los signif de este último, sobre su dicc. de tema, y voy insertando (en un arreglo de n posiciones) tantas veces k como diga la 2da componente de su signif en el dicc. de tema.

Recorrer un dicc. de n elem. sobre AVL lleva $n \log(n)$. Como va a haber tantos ~~elem~~ elem como palabras, en este caso lleva $b \log(b)$.

Recorrer un dicc de ~~tema~~ lleva $k \cdot |q|$, donde k son las ~~tema~~ claves y $|q|$ la long max de la clave.

En este caso llevaría $k|d|$ para el de discos y $k|T|$ para el de temas.

~~Como en total hay b mp3 recorridos en los dicc. y los dicc están arbolados~~

Como están arbolados, esto va a llevar $b \log(b) |T| |d|$.

Sin embargo como tengo que expandir las tuplas del dicc de temas, voy a tardar $n \cdot \log(b) |T| |d|$.

nro de elem en el arreglo orig.